Theoretical Computer Science 459 (2012) 100–112

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

The binary identification problem for weighted trees☆

Ferdinando Cicalese[a], Tobias Jacobs[b], Eduardo Laber[c,*], Caio Valentim[c]

[a] University of Salerno, Italy
[b] NEC Laboratories Europe, Germany
[c] PUC, Rio de Janeiro, Brazil

ARTICLE INFO

Article history:
Received 20 September 2011
Received in revised form 2 April 2012
Accepted 18 June 2012
Communicated by G. Italiano

ABSTRACT

The Binary Identification Problem for weighted trees asks for the minimum cost strategy (decision tree) for identifying a vertex in an edge weighted tree via testing edges. Each edge has assigned a different cost, to be paid for testing it. Testing an edge e reveals in which component of $T - e$ lies the vertex to be identified. We give a complete characterization of the computational complexity of this problem with respect to both tree diameter and degree. In particular, we show that it is strongly NP-hard to compute a minimum cost decision tree for weighted trees of diameter at least 6, and for trees having degree three or more. For trees of diameter five or less, we give a polynomial time algorithm. Moreover, for the degree 2 case, we significantly improve the straightforward $O(n^3)$ dynamic programming approach, and provide an $O(n^2)$ time algorithm. Finally, this work contains the first approximate decision tree construction algorithm that breaks the barrier of factor $\log n$.

1. Introduction

Given a set of objects $U = \{u_1, \ldots, u_n\}$, a set of tests $\{t_1, \ldots, t_m\}$, with $t_i \subseteq U$, a cost function $c : \{t_1, \ldots, t_m\} \mapsto \mathbb{R}^+$ and a 'hidden' marked object, the Binary Identification Problem (BIP) consists of defining a strategy (decision tree) for adaptively selecting a sequence of tests that minimizes the cost of identifying the marked object in the worst case [8]. A test t incurs a cost $c(t)$ and allows to determine whether the marked object is in the set t or in $U \setminus t$. The BIP is an $\mathcal{NP}$-Complete problem that does not admit an $o(\log n)$-approximation unless $\mathcal{P} = \mathcal{NP}$ [20,13]. On the other hand, a simple greedy algorithm attains an $O(\log n)$-approximation [21].

Here, we study the version of the BIP in which the underlying space of objects and tests can be represented by a weighted tree. By a weighted tree we understand a pair $(T, \mathbf{c})$ where T is a tree and $\mathbf{c}$ is a cost assignment to the edges $E(T)$ of T, i.e., $\mathbf{c} : e \in E(T) \mapsto c(e) \in \mathbb{R}_0^+$.

A decision tree for a weighted tree $(T, \mathbf{c})$ is a binary tree recursively defined as follows: if the tree T has only one vertex, then the decision tree is a single leaf labeled with the only vertex in T. If T has at least one edge, a decision tree for T has its root r labeled with one edge $e = \{u, v\}$ in T, and the subtrees rooted at the children of r are decision trees for the connected components T_u and T_v of $T - e$.

For the sake of distinguishing between the input tree and the decision tree, we shall reserve the term *node* to the decision tree and the term *vertex* to the input tree.

A decision tree D for $(T, \mathbf{c})$ naturally defines a strategy for identifying an initially unknown vertex x from T via edge queries. If node w of D is labeled with the edge $e = \{u, v\}$ of T, we map w to the question "Is x in T_u or in T_v?", where T_u

☆ This paper has appeared in preliminary form in the proceedings of WADS 2011.

* Corresponding author. Tel.: +55 2199113371.
E-mail address: laber@inf.puc-rio.br (E. Laber).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.06.023

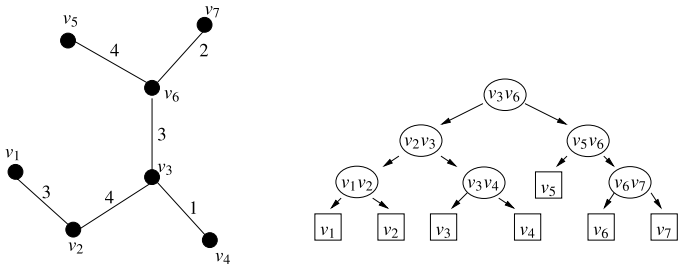

Fig. 1. An edge weighted tree T (left hand side) and a decision tree D for it (right hand side). For example, identifying the vertex v_3 is done by queries to the edges $\{v_3, v_6\}$, $\{v_2, v_3\}$, and $\{v_3, v_4\}$. This identification process has a cost of $3 + 4 + 1 = 8$. The maximum cost of 10 has to be payed for the identification of v_2, and this worst case cost defines the cost of the depicted decision tree.

(resp. T_v) denotes the component of $T - e$ which contains u (resp. v). The search strategy now consists in starting with the query at the root of D and then recursively continuing with the subtree being a decision tree for the component indicated in the answer. Accordingly, each leaf ℓ of D is then labeled with the vertex of T uniquely identified by the sequence of questions and answers corresponding to the path from the root of D to ℓ.

The cost of a decision tree D for T is 0 if D consists of just one leaf (i.e., T has only one vertex), and otherwise it is the cost of the edge in the root of D plus the maximum of the costs of the decision trees rooted at the children of the root of D, in formulas

$$cost(D) = c(root(D)) + \max\{cost(D_L), cost(D_R)\},$$

where D_L and D_R are the decision trees rooted at the left and right child of the root of D, respectively.

We also define the cost of searching a single vertex $u \in T$ according to D as the sum of the costs of the edges labeling the nodes in the path from the root of D to the leaf labeled with u. With this definition, we have that the cost of D is equal to the maximum among the search costs of the vertices from T according to D.

Given a weighted tree $(T, \mathbf{c})$ the Binary Identification Problem asks for the decision tree for T of minimum cost. We refer to Fig. 1 for an example.

Our results. We provide a complete characterization of the complexity of the Binary Identification Problem for weighted trees in terms of both the maximum degree and the diameter of the input tree. We show that the problem is strongly NP-hard already for bounded instances of diameter 6 or maximum degree 3. Both thresholds are tight. In fact, we give a polynomial time algorithm for instances of bounded diameter at most 5. We reserve special attention to the case of instances of maximum degree 2 (simple paths). It is easy to see that for such instances, a natural dynamic programming approach results in an $O(n^3)$ algorithm for building an optimal decision tree, and, to the best of our knowledge, no algorithm with better asymptotic was known prior to this paper. We present a non-trivial DP based algorithm which provides the optimal decision tree in $O(n^2)$ time. Such a speed up has been obtained for problems with the same flavor by employing the Knuth–Yao technique [11,19]. However, this technique cannot be directly applied to the problem considered here as we discuss in Section 4.

Finally, for general trees, we provide an $O(\log n/\log\log\log n)$-approximation algorithm. Although this result is not a significant improvement, in numerical terms, over the existing $O(\log n)$ approximation [6], it is interesting as it shows a sharp separation in the complexity picture of the binary identification problem with costs. This is because the general BIP (not restricted to tree instances), even with uniform weights, does not admit an $o(\log n)$-approximation unless $P = NP$ [13].

Related work. The binary identification problem (BIP) for unweighted trees has been extensively studied in the context of searching and edge ranking [9,5,14,1,17,18,4,16]. The edge ranking problem and its connection to the problem studied here is precisely explained later when we discuss some applications. Linear time algorithms that construct an optimal decision tree for unweighted trees are presented in [14,17].

The BIP for weighted trees was first studied by Dereniowski [6] in the context of edge ranking. In this initial paper, the problem was defined and proved to be NP-complete already for the class of instances of diameter at most 10. In addition, an $O(\log n)$ approximation algorithm was also provided. In fact, the $O(\log n)$ approximation can be attained for the general version of the BIP (not restricted to tree instances), via a simple greedy procedure [3].

When the weighted tree is a path, the BIP is equivalent to the problem of searching in an ordered array with costs depending on the position probed. A natural DP approach solves this problem in $O(n^3)$ time. A linear time algorithm with constant approximation factor is presented in [12]. In [2], Charikar *et al.* consider this problem from a competitive analysis perspective.

Applications. The BIP is a basic problem in computer science and has applications in many different scenarios.

The BIP for (weighted) trees arises when one has to identify the faulty component of a system. As an example, a system is represented by a network (in our case a tree) and its faulty component (vertex) has to be found. Different points of the network might require more or less expensive operations for the inspection. Inspecting one spot (edge) in the network reveals only directional information about the location of the failure w.r.t. the inspected point. One such problem is

described, e.g., in [15] as searching for holes in an oil pipeline. In [18], the problem of finding a bug in a software application is mentioned.

As already pointed out, the BIP for trees is equivalent to the edge ranking problem for trees. An edge ranking of T is an assignment to each edge e of T of an integer $r(e)$ (the rank of e) s.t. for any two edges $e_1, e_2 \in T$ if $r(e_1) = r(e_2)$, then the path connecting e_1 and e_2 contains an edge e with $r(e) > r(e_1)$. The cost of an edge ranking of a weighted tree $(T, \mathbf{c})$, denoted by $\text{rankcost}(T, \mathbf{c})$ is defined as follows: if T has only one vertex, then $\text{rankcost}(T, \mathbf{c}) = 0$. Otherwise, $\text{rankcost}(T, \mathbf{c}) = c(e^*) + \max\{\text{rankcost}(T_u, \mathbf{c}), \text{rankcost}(T_v, \mathbf{c})\}$, where $e^* = \{u, v\}$ is the edge with maximum rank in T and T_u (resp. T_v) is the connected component of $T - e$ that contains u (resp. v). Given a weighted tree $(T, \mathbf{c})$ the edge ranking problem asks for the minimum cost ranking. The equivalence to the decision tree problem is easily seen (see also [7]).

The edge ranking problem arises in the context of multi-part product assembly [6,10]. Assume that each edge represents the operation of assembling two parts of a product and the weight of an edge represents the time necessary to complete the corresponding assembly operation. Each product part can only participate in one assembly operation at a time, which means that, whenever two edges share an endpoint, the corresponding operations are dependent and cannot be performed simultaneously. An edge ranking provides a scheduling of the assembly operations with the guarantee that only independent operations are scheduled simultaneously. Moreover, the cost of the edge ranking is the total time necessary for completely assembling the product.

Paper organization. Our paper is organized as follows. In Section 2, we present the hardness proofs. In Section 3, we present a simple polynomial time algorithm for instances of diameter at most 5. In Section 4, we give an $O(n^2)$ time algorithm for path instances. In Section 5, we present the $O(\log / \log\log\log n)$ approximation algorithm for general weighted trees.

2. Proofs of strong NP-hardness

Our proofs of NP-hardness proceed in two steps. In the first step, we reduce from a certain scheduling problem, which we call *Flexible Machine Scheduling (FMS)*. This problem is reduced both to the BIP on weighted trees of diameter 6 and to the problem on degree 3 trees. Both reductions have the property that the edge costs of the resulting weighted tree instances are polynomial in the processing times and deadlines of the scheduling instance reduced from. In the second step, we reduce Problem 3SAT to FMS and thereby show strong NP-hardness of that scheduling problem.

The Flexible Machine Scheduling problem is defined as follows:. We are given k pairwise disjoint sets of jobs $S_1, \ldots, S_k$. Each job J from one of those sets is characterized by its length $l(J) \in \mathbb{R}_0^+$ and deadline $d(J) \in \mathbb{R}_0^+$. Furthermore, each job set S_i has a so-called *setup time* $s_i \in \mathbb{R}_0^+$. Initially, the jobs have to be scheduled on a single machine M. However, at any point of time t we can open an extra machine M_i, $1 \leq i \leq k$. After machine M_i has been opened, the remaining jobs from set S_i can be processed on it. More precisely, M_i can only process jobs from S_i, and these jobs may not be processed on any other machine after M_i has been opened. Opening machine M_i takes time s_i, which means that M cannot process any job during the time interval $(t, t + s_i)$, and also the new machine M_i becomes only available at time $t + s_i$. Opening other machines is also not possible during that time interval. This implies that the setup time intervals of any two machines M_i, M_j can never overlap. We are interested in the problem of deciding whether for a given FMS instance there exists a feasible solution, i.e. one where each job is completely processed before its deadline is reached. We shall note that preemption is not allowed in our model.

In the following we show how to reduce FMS to the bounded diameter case of BIP on weighted trees. After that, we will explain how to modify the reduction in order to obtain bounded degree instances.

Reduction of FMS to bounded diameter instances. It will be comfortable to talk of a decision tree in terms of the search strategy it defines. Recall that, in this perspective, we interpret the edge labels of the nodes of the decision tree as queries. Also, by the search cost of a vertex v in the input tree we mean the sum of costs of the edges queried in the decision tree on the path from the root to the leaf labeled with v. We will call such a path the search path to/of v. We also say that this path isolates v.

We first state an observation that plays a central role in this and the subsequent section.

Lemma 1. *Let l be a leaf in T and let u be its adjacent vertex in T. Then, the search cost of l is not larger than the search cost of u in every decision tree for T.*

Proof. In every decision tree for T, the search path to l coincides with the search path to u until these two vertices are separated from each other via a query to $\{u, l\}$. After that query the leaf l is already isolated. □

Lemma 2. *Let $(T, \mathbf{c})$ be a weighted tree with at least one internal vertex. Let l be a leaf of T and u be the neighbor of l. For any decision tree D for $(T, \mathbf{c})$ such that the edge $e = \{u, l\}$ is queried at the root of D there is another decision tree D' where e is the last query on the search path of u and $\text{cost}(D') \leq \text{cost}(D)$.*

Proof. By definition, one of the children of the root of D is a leaf labeled with l. Let D_1 be the subtree rooted at the other child of the root of D. By definition D_1 is a decision tree for $(T - \{l\}, \mathbf{c})$.

Let D' be the decision tree obtained by substituting in D_1 the leaf labeled with u with a query to $e = \{l, u\}$ and adding as children of this new node the leaves labeled u and l. It is not hard to see that D' is a decision tree for $(T, \mathbf{c})$. In fact, any vertex

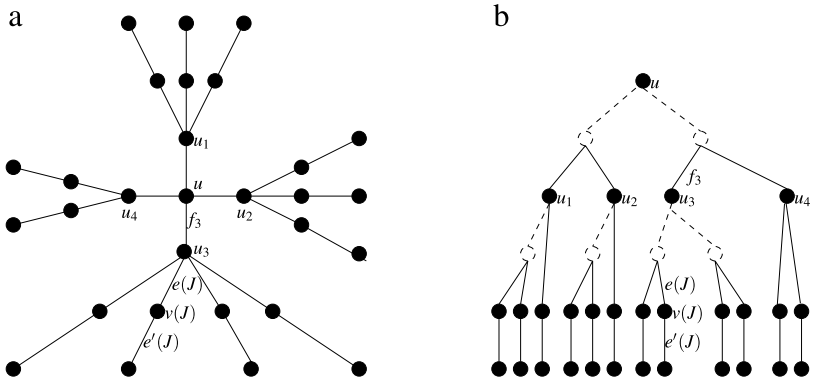

Fig. 2. (a) A diameter 6 tree T obtained from the reduction of an instance of FMS. Here J is a job in S_3. (b) The degree 3 realization T' of the same instance. The dashed parts are the elements that do not occur in T.

other than u and l is separated from all the other as it was in D_1. Moreover, u and l are separated by the query we added to the search path of u.

Regarding the cost, we have that

1. for each $v \in T - \{u, l\}$, the search cost $cost'_D(v)$ of v in D' is not larger than the search cost $cost_D(v)$ of v in D, since in D' the search path to v does not contain the query to e, which, instead, it did in D.

2. $cost_{D'}(l) = cost_{D'}(u) = cost_D(u) \leq cost(D)$, where the first equality follows from the fact that u and l are siblings in D'. The inequality follows from the observation that the search path of u in D' is the same as its search path in D but for the fact that the position of the query to e is changed.

From 1. and 2. it immediately follows that $cost(D') \leq cost(D)$. □

Corollary 1. *For any instance of the weighted tree problem, there is an optimal search strategy where every leaf l is separated from its neighbor u only after u has been separated from all its non-leaf neighbors.*

Proof. Let $(T, \mathbf{c})$ be an instance of the weighted tree problem and let D be an optimal decision tree for this instance. We say that a leaf $l \in T$ is *bad located* in D if l is separated from its neighbor u in D before u has been separated from all its non-leaf neighbors.

Let D^* be an optimal decision tree for $(T, \mathbf{c})$ with the minimum number of bad located leaves. If D^* has no bad located leaves then D^* satisfies the required property. Otherwise, let l be a bad located leaf in D^* with maximum depth and let u be the neighbor of l in T. Then, we can apply Lemma 2 to the subtree of D^* rooted at l, u obtaining a new optimal decision tree where the edge $\{l, u\}$ is queried last on u's search path. However, this new optimal tree has less bad located leaves than D^*, which contradicts the minimality of D^* and establishes the corollary. □

Let I be an instance of FMS, determined by the job sets $S_1, \ldots, S_k$ and the corresponding setup times $s_1, \ldots, s_k$. Let $S = S_1 \cup \cdots \cup S_k$. For each job $J \in S$, the tree T contains two edges $e(J)$, $e'(J)$. We set the cost of $e(J)$ to the length $l(J)$ of J. The cost of $e'(J)$ is $A - d(J)$, where $A \in \mathbb{R}$ is a large constant, depending on instance I, whose exact value will be determined later. The edges $e(J)$ and $e'(J)$ have a common endpoint denoted $v(J)$. The other endpoint of $e'(J)$ is a leaf.

Before continuing with the description of $(T, \mathbf{c})$, we give some intuition about the idea of the reduction. I will be reduced to the problem of deciding whether there exists a search strategy for $(T, \mathbf{c})$ where the search cost of any vertex is no more than A. Observe that, in order to isolate the vertex $v(J)$, the edge $e'(J)$ has to be queried. This means that the total cost of all other queries on the search path of vertex $v(J)$ must not exceed $d(J)$.

For each job set S_i, $i = 1, \ldots, k$, the tree T contains a vertex u_i, which serves as the common endpoint of all edges $e(J)$, so that $e(J)$ connects u_i with $v(J)$ for each $J \in S_i$. In addition u_i has one further incident edge f_i, whose cost is set to the setup time s_i. The construction of T is completed by letting $f_1, \ldots, f_k$ share the common endpoint u, so that f_i connects u with u_i for $i = 1, \ldots, k$. Clearly, T has diameter 6, see Fig. 2(a) for an example.

Intuitively, a query to edge f_i will correspond to opening machine M_i in the FMS instance. This causes additional search cost s_i to all vertices that have not been separated from the central vertex u.

Lemma 3. *Fix $A = \sum_{i=1}^{k} c(f_i) + \sum_{J \in S} d(J)$. There is a feasible solution to I if and only if there is a decision tree for $(T, \mathbf{c})$ of cost not larger than A.*

Proof. "⇒" Assume that there is a feasible schedule for I. If some machines are not opened during the execution of the schedule, we can equivalently assume that these machines are opened after all jobs have been processed. We can also assume that the only time machine M is idle is during the setup of some other machine, and there is no idle time between jobs on any other machine. A feasible schedule with that property can easily be constructed from an arbitrary feasible schedule.

A search strategy for $(T, \mathbf{c})$ is constructed from the schedule by interpreting the assignments to machine M as the search path to vertex u. Assigning job J to M corresponds to a query to edge $e(J)$, and opening machine M_i corresponds to a query

to edge f_i. For each $J \in S$, we query the edge $e'(J)$ right after $e(J)$ has been queried. We have to show that the cost to each node of T is upper bounded by A.

The cost of the search path to u equals the point in time when the last job has been finished on M and the last machine has been opened. Thus, it is upper bounded by $\sum_{i=1}^{k} c(f_i) + \sum_{J\in S} l(J) \leq \sum_{i=1}^{k} c(f_i) + \sum_{J\in S} d(J) = A$.

For $i = 1, \ldots, k$, vertex u_i shares its search path with u until edge f_i is queried. After that, its search path is represented by the order of the jobs on machine M_i, where job $J \in S_i$ corresponds to edge $e(J)$. The cost of the search path to u_i therefore corresponds to the point in time when the last job from S_i has been completed and the machine M_i has been opened. Thus, it is also upper bounded by $\sum_{i=1}^{k} c(f_i) + \sum_{J\in S} l(J) \leq A$.

The search cost of $v(J)$ is equal to the completion time of job J plus the cost of edge $e'(J)$. As the schedule is feasible, this search cost is at most $d(J) + c(e'(J)) = A$. The search path to the leaf incident to $e'(J)$ has the same cost as $v(J)$.

"⇐" Assume that there is a search strategy for $(T, \mathbf{c})$ where each vertex has search cost A or less. Because of Corollary 1 we can assume that $e(J)$ is queried before $e'(J)$ for any $J \in S$.

Under this assumption, a schedule for I can be directly constructed from the search strategy. The schedule with respect to machine M processes jobs J in exactly the order of the search path to u, where a query to $e(J)$ corresponds to processing of job J, and a query to an edge f_i is translated into opening machine M_i. After M_i has been opened, it processes the unprocessed jobs J from S_i in the order in which the corresponding edges $e(J)$ are queried after the query to f_i. This way, we achieve that the completion time of job J is by $c(e'(J)) = A - d(J)$ smaller than the search cost of $v(J)$. Since that search cost is no more than A by assumption, J is completed by time $d(J)$. □

Reduction of FMS to bounded degree instances. The tree T has diameter 6, but its degree is unbounded. For constructing a bounded degree tree instead, we need to replace the star structure of T with a binary tree structure. Construct a binary tree rooted at u, having k leaves $u_1, \ldots, u_k$. The edges $f_1, \ldots f_k$ are the edges of that binary tree that end in $u_1, \ldots, u_k$, respectively. Note that the binary tree can have an arbitrarily chosen topology as long as it respects the imposed conditions. Now enhance the tree constructed so far by making u_i the root of a binary tree having $|S_i|$ leaves $v(J), J \in S_i$, for $i = 1, \ldots, k$. For $J \in S_i$, the edge incident to $v(J)$ is $e(J)$. Finally, for $J \in S$ add a further outgoing edge to $v(J)$, namely, the edge $e'(J)$. The other end points of the edges $e'(J), J \in S$, are the final leaves of the constructed tree T'. The edges $e(J)$, $e'(J)$ and f_i have the same costs as before, except that we need A to have a different value A'. An example of the resulting tree T' can be found in Fig. 2(b).

Let E' be the set of all edges that do not occur in the diameter 6 realization of T. We need to ensure that no edge from E' appears on the search path to some $v(J)$. This is achieved by making them expensive: we assign cost $c' = \max\{d(J), J \in S\}+1$ to them, which implies that no search strategy querying an edge from E' during the search for a $v(J)$ can reach the cost bound A. As we still need the search costs of the $e'(J)$s to be dominating the other vertices, we set $A' = |T'|c' + \max\{d(J), J \in S\} + 1$ here.

As only the search paths to the vertices $e'(J)$ are relevant for the cost of an optimal search strategy for the resulting tree T' and on these search paths there appear only edges that are also present in T, any optimal search strategy for T' can be directly translated into a search strategy for T and vice versa. Therefore, there is a cost A search strategy for T if and only if there is a cost A' strategy for T'.

Strong Hardness of FMS. We show by reduction from 3SAT that Problem FMS is strongly NP-hard.

Definition 1 (*3SAT*)**.** Given a set of m clauses $C_1, \ldots, C_m$ over a set of n boolean variables $x_1, \ldots, x_n$, where each clause is a disjunction of exactly three literals, decide whether there is an assignment to the variables such that each clause is satisfied.

Let $C_1, \ldots, C_m$ be an instance of 3SAT with variables $x_1, \ldots, x_n$. We show how to construct an equivalent instance I of FMS.

Instance I consists of $2n$ sets of jobs $S_1, \ldots, S_n, \bar{S}_1, \ldots, \bar{S}_n$ which correspond to the literals $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$, respectively. The setup time of all job sets is 2. For $i = 1, \ldots, n$, S_i contains a job J_i with processing time 1 and deadline $5(i-1)+3$. Furthermore, S_i contains a job K_i with processing time 2 and deadline $5i$. The set $\bar{S}_i$ contains a pair of jobs $\bar{J}_i$, $\bar{K}_i$ with the same characteristics as J_i, K_i. Those $4n$ jobs will enforce that at time $5n$, for any $i = 1, \ldots, n$, the schedule has opened exactly one of the two machines associated with S_i and $\bar{S}_i$. Note that the construction so far is independent of the clauses.

Now, for $j = 1, \ldots, m$, add jobs $L_{j1}, \ldots, L_{jn}, \bar{L}_{j1}, \ldots, \bar{L}_{jn}$ to $S_1, \ldots, S_n, \bar{S}_1, \ldots, \bar{S}_n$, respectively, all having processing time 1. The jobs added to the three sets corresponding (in the sense defined in the previous paragraph) to the literals in C_j have deadline $5n + (j-1)n + 2$, while the deadline of all $2n - 3$ other jobs is $5n + jn$. The idea of this construction is that we need at least one machine corresponding to a literal in C_j to be open, because we cannot process all three jobs with deadline $5n + (j-1)n + 2$ on machine M.

As a toy example, consider an instance of 3SAT with three variables x_1, x_2, x_3 and two clauses $C_1 = (x_1 \vee \bar{x}_2 \vee x_3)$ and $C_2 = (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$. The resulting FMS instance has six job sets $S_1, \bar{S}_1, S_2, \bar{S}_2, S_3, \bar{S}_3$. Both S_1 and $\bar{S}_1$ contain a job J_1 (resp. $\bar{J}_1$) with processing time 1 and deadline 3 and a job K_1 (resp. $\bar{K}_1$) with processing time 2 and deadline 5. The total processing time of these jobs is 6, so they cannot all be processed on machine M, and either the machine corresponding to S_1 or the one corresponding to $\bar{S}_1$ has to be opened. Opening both machines is not possible, because then either J_1 or $\bar{J}_1$ would miss its deadline 3. It is not hard to see that the only two feasible options are to open the machine for S_1 at time 0 and then subsequently process $\bar{J}_1$ and $\bar{K}_1$ on M, or to open the machine for $\bar{S}_1$ at time zero and then subsequently process J_1 and K_1

on M. In either case, machine M is kept busy until exactly time 5. After that, the same reasoning shows that between time 5 and time 10 the jobs J_2, K_2, $\bar{J}_2$, and $\bar{K}_2$ have to be processed, and for achieving that either the machine for S_2 or the machine for $\bar{S}_2$ has to be opened at time 5. Finally, between time 10 and 15 the same reasoning holds for the corresponding four jobs in S_3 and $\bar{S}_3$.

At time 15 the phase begins where the jobs L_{ji}, $\bar{L}_{ji}$, $j = 1, \ldots, 3$, $i = 1, 2$ have their deadlines. Recall that for $j = 1, 2, 3$, L_{j1} and L_{j2} belong to set C_j, while $\bar{L}_{j1}$ and $\bar{L}_{j2}$ are members of $\bar{C}_j$. The deadlines of the jobs $(L_{11}, \bar{L}_{11})$, $(L_{21}, \bar{L}_{21})$, $(L_{31}, \bar{L}_{31})$ are all before time 18. For each of these three job pairs, one member has to be processed on M while the other is processed on its own machine that has been opened before time 15. As $C_1 = (x_1 \vee \bar{x}_2 \vee x_3)$, the exact deadline of L_{11}, $\bar{L}_{21}$, and L_{31} is 17, while the remaining jobs $\bar{L}_{11}$, L_{21}, and $\bar{L}_{31}$ have deadline 18. So if before time 15 neither of the machines corresponding to S_1, $\bar{S}_2$, S_3 has been selected to be opened, the three deadline 17 jobs need to be processed on machine M between time 15 and time 17, which is not feasible. This selection of opened machines corresponds to variable configuration $(x_1, x_2, x_3) = (0, 1, 0)$ in the 3SAT instance, which effectuates that clause C_1 is not satisfied. In contrast, any selection of open machines that leads to satisfaction of C_1 effectuates that one of the three jobs to be processed on M has deadline 18 and the six jobs $(L_{11}, \bar{L}_{11})$, $(L_{21}, \bar{L}_{21})$, $(L_{31}, \bar{L}_{31})$ can be processed before their deadlines. The same argumentation holds for the six jobs corresponding to clause C_2, which have to be scheduled between time 18 and time 21.

Lemma 4. *The 3SAT instance has a solution if and only if a feasible schedule exists for I.*

Proof. "$\Rightarrow$" Let there be a satisfying assignment for the 3SAT instance. We construct a schedule for I as follows. For $i = 1, \ldots, n$, if the assignment to x_i is positive, then open the machine associated with S_i; process J_i and K_i in this machine and process $\bar{J}_i$ and $\bar{K}_i$ on the initial machine M. If the assignment to x_i is negative, then open the machine associated with $\bar{S}_i$; process $\bar{J}_i$ and $\bar{K}_i$ in this machine and J_i and K_i on the initial machine M.

A simple argument of induction shows that this effectuates that J_i and $\bar{J}_i$ are completed exactly at the time of their deadline $5(i-1)+3$, and also the processing of K_i and $\bar{K}_i$ finishes by their deadline $5i$.

After that, all remaining jobs are scheduled in ascending order of their deadlines on the respective machines, breaking ties arbitrarily. It is not hard to see that the opened machines can process their jobs on time without any problem.

Thus, it remains to show what happens with machine M. For $j = 1, \ldots, m$, machine M has to process n jobs out of $L_{j1}, \ldots, L_{jn}, \bar{L}_{j1}, \ldots, \bar{L}_{jn}$. Simple induction shows that these n jobs are processed between time $5n + (j-1)n$ and $5n + jn$, which means that all jobs with deadline $5n + jn$ processed by M finish on time. As clause C_j is satisfied by the assignment, at least one of the three jobs with deadline $5n + (j-1)n + 2$ is processed by an opened machine, and therefore there are at most two jobs with deadline $5n + (j-1)n + 2$ that are processed by M; those can be processed in the time interval $[5n + (j-1)n, 5n + (j-1)n + 2]$ without incurring a delay.

"$\Leftarrow$" Assume that there is a feasible schedule for I. We show by induction that for $i = 1, \ldots, n$, the jobs J_i, K_i and $\bar{J}_i$, $\bar{K}_i$ have to be processed between time $5(i-1)$ and $5i$, and neither the machine associated with S_i nor the one associated with $\bar{S}_i$ has been opened before time $5(i-1)$.

Assume that this has been shown for all $i' < i$, including the base case $i = 1$. Then we have a time interval of length 5 for processing J_i, K_i and $\bar{J}_i$, $\bar{K}_i$. Those jobs have a total processing time of 6, so either the machine associated with S_i or the one associated with $\bar{S}_i$ must be opened. However, we cannot open both machines, because then either J_i or $\bar{J}_i$ will be delayed. If we open one of the machines, say the one associated with S_i, then there is a time slot of length 3 left on M for processing $\bar{J}_i$ and $\bar{K}_i$, which means that we cannot open any other machine until time $5i$, which establishes the induction claim.

For $j = 1, \ldots, m$, each of the n job sets whose machine has not been opened before time $5n$ contains one job with deadline either $5n + jn$ or $5n + (j-1)n + 2$. A similar argument of induction as above shows that those n jobs have to be scheduled between time $5n + (j-1)n$ and $5n + jn$, and no further machine can be opened during the whole time interval from $5n$ to $5n + mn$. Furthermore, at least one of the three jobs with deadline $5n + (j-1)n + 2$ must belong to a set whose machine has been opened before time $5n$, because otherwise we need to schedule three length 1 jobs on M in the time interval $[5n + (j-1)n, 5n + (j-1)n + 2]$ of length 2, which is impossible. As this property holds for $i = 1, \ldots, m$, the set of opened machines corresponds to a satisfying assignment for the 3SAT instance. □

Theorem 1. *The BIP on weighted trees is strongly NP-hard on instances of diameter* 6*. The same complexity holds for degree* 3 *instances.*

Proof. The reduction from 3SAT, whose correctness is proven in Lemma 4, shows that FMS is strongly NP-hard. We have given a fully polynomial time reduction of FMS to instances of BIP on trees with diameter 6, and on trees of degree 3. The correctness of the reduction has been proven by Lemma 3, and it shows that from a pseudopolynomial algorithm for the tree searching problem one could construct one for FMS, which is not possible unless $P = NP$. □

3. A polynomial time algorithm for diameter 5 instances

In this section we show that with respect to the diameter the threshold of 6 in the hardness result of the previous section is tight. We provide a polynomial time algorithm for instances of diameter not larger than 5. The following lemma allows us to assume that in the tree under consideration each vertex has at most one leaf as neighbor, and Lemma 6 allows us to concentrate on a certain class of search strategies for such trees.

For any vertex u in T, let $L(u)$ be the set of leaves adjacent to u.

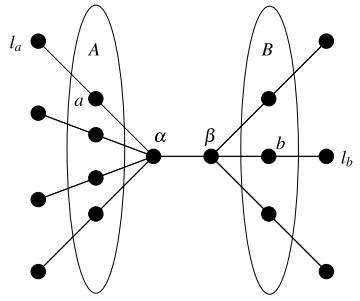

Fig. 3. An instance of a diameter 5 tree.

Lemma 5. *For some internal vertex u of T, let $L(u)$ be the set of its leaf neighbors. Let $(\tilde{T}, \tilde{\mathbf{c}})$ be obtained from $(T, \mathbf{c})$ by replacing the leaves from $L(u)$ and the edges $\{(u, v) \mid v \in L(u)\}$ with a single leaf l_u and edge $\{u, l_u\}$ with cost $\tilde{c}(\{u, l_u\}) = \sum_{v \in L(u)} c(\{u, v\})$, while for all other edges, $\tilde{\mathbf{c}}$ and $\mathbf{c}$ coincide. Then $(T, \mathbf{c})$ and $(\tilde{T}, \tilde{\mathbf{c}})$ are equivalent problem instances, i.e. an optimal solution to $(T, \mathbf{c})$ can be transformed in polynomial time into an optimal solution to $(\tilde{T}, \tilde{\mathbf{c}})$ and vice versa.*

Proof. Let D an optimal search strategy for $(T, \mathbf{c})$ satisfying the property formulated in Corollary 1. We can easily transform D into a search strategy $\tilde{D}$ for $(\tilde{T}, \tilde{\mathbf{c}})$: At the point in time when D would successively query $\{u, v\}$, $v \in L(u)$, $\tilde{D}$ queries edge $\{u, l_u\}$.

A search strategy $\tilde{D}$ for $(\tilde{T}, \tilde{\mathbf{c}})$ can be transformed into a search strategy D for T in a similar way: D successively queries the edges between u and the leaves from $L(u)$ at the point in time where $\tilde{D}$ would query $\{u, l_u\}$.

The search strategies $\tilde{D}$ and D have the same worst-case search cost, because the individual cost of each non-leaf vertex is the same under both strategies, and the search costs of the leaves do not matter because of Lemma 1. □

Under the assumption of the preceding lemma, any tree with diameter exactly 5 can be described as follows. There is a "central" edge $\{\alpha, \beta\}$, and α and β are connected to the set of vertices $\{\beta\} \cup A$ and $\{\alpha\} \cup B$, respectively. Each $a \in A$ is connected to a leaf l_a, and each $b \in B$ is connected to a leaf l_b. Although there exist diameter 5 trees where some a does not have a neighbor besides α, we can assume in that case that the edge $\{a, l_a\}$ has cost zero. See Fig. 3 for an example tree with $|A| = 4$ and $|B| = 3$.

The edges $\{a, l_a\}$, $a \in A$ and $\{b, l_b\}$, $b \in B$ are called *outer edges*, and the edges incident to α and β, except edge $\{\alpha, \beta\}$, are called *inner edges*. From Corollary 1 we know that any outer edge $\{a, l_a\}$ or $\{b, l_b\}$ can be assumed to be queried after the query to the respective inner edge $\{\alpha, a\}$ or $\{\beta, b\}$. We therefore only need to reason about the optimal strategy for querying the edges incident to α and β. Furthermore, we can ignore the search costs of the leaves, because they are dominated by the search costs of their respective neighbors. The following lemma shows that we can restrict ourselves to certain ordered strategies.

Lemma 6. *There is an optimal solution that is* ordered, *i.e.*

(a) *the inner edges (α, a), $a \in A$, are queried in the order of the non-increasing cost of their respective outer edges $\{a, l_a\}$. The same holds for the edges $\{\beta, b\}$, $b \in B$.*
(b) *the inner edges queried before $\{\alpha, \beta\}$ are queried in the order of the non-increasing cost of their respective outer edges.*

Proof. Assume that we are given an optimal solution. We show how to modify it in order to obtain an optimal solution satisfying properties (a) and (b).

We first show how to establish property (b). Assume that, before the query to $\{\alpha, \beta\}$, a query to some inner edge, say $\{\alpha, a\}$ is followed by the query to another inner edge, say $\{\beta, b\}$, with $c(\{a, l_a\}) < c(\{b, l_b\})$. Let c_0 be the search costs payed before the query to $\{\alpha, a\}$, i.e., the search cost of a is $c_0 + c(\{\alpha, a\}) + c(\{a, l_a\})$. The search cost of b is then $c_0 + c(\{\alpha, a\}) + c(\{\beta, b\}) + c(\{b, l_b\})$. If one reverses the query order of $\{\alpha, a\}$ and $\{\beta, b\}$, then the search cost of b decreases and the search cost of a becomes $c_0 + c(\{\beta, b\}) + c(\{\alpha, a\}) + c(\{a, l_a\})$, which not larger than the former search cost of b. The search cost of all other vertices remains the same, and thus the obtained solution is optimal. That kind of interchange operation can be performed as long as property (b) is not satisfied, and after a finite number of iterations the property will hold.

The interchange operation can also be used to achieve that all queries to inner edges $\{\alpha, a\}$, $a \in A$ performed *after* the query to $\{\alpha, \beta\}$ are ordered by non-increasing cost of the respective outer edge. The same can be achieved for the queries to $\{\beta, b\}$, $b \in B$ performed after the query to $\{\alpha, \beta\}$. We call this property (c) and remark that (c) is a weaker version of (a).

Assume that properties (b) and (c) are satisfied. If also (a) holds, we are done. Otherwise assume w.l.o.g. that the inner edges (α, a), $a \in A$ are not queried in the desired order. Let $\{\alpha, a\}$ be the last query to an edge between α and an element of A that is performed before the query to $\{\alpha, \beta\}$. Let further $\{\alpha, a'\}$, $a' \in A$, be the first query of that kind performed after the query to $\{\alpha, \beta\}$. It holds that $c(\{a, l_a\}) < c(\{a', l_{a'}\})$, due to the assumption that property (a) is violated. Let $B' \subseteq B$ be the

set of vertices b where query $\{\beta, b\}$ is performed between the queries to $\{\alpha, a\}$ and to $\{\alpha, \beta\}$. As all queries before $\{\alpha, \beta\}$ are ordered, it holds that

$$c(\{b, l_b\}) \leq c(\{a, l_a\}) < c(\{a', l_{a'}\}) \quad \text{for all } b \in B' . \tag{1}$$

Let c_0 be the search cost that is payed before the query to $\{\alpha, a\}$. The search cost of a' is

$$c_0 + c(\{\alpha, a\}) + \sum_{b \in B'} c(\{\beta, b\}) + c(\{\alpha, \beta\}) + c(\{\alpha, a'\}) + c(\{a', l_{a'}\}) . \tag{2}$$

We modify the search strategy by placing the query to $\{\alpha, \beta\}$ immediately before $\{\alpha, a\}$. Now the subsequent queries to a and $b \in B'$ take place in different branches of the search tree. The only search paths that can become more expensive by this modification are the ones leading to a and to $b \in B'$. The new search cost of a is

$$c_o + c(\{\alpha, \beta\}) + c(\{\alpha, a\}) + c(\{a, l_a\}) ,$$

and the new search cost of any vertex in B' is at most

$$c_o + c(\{\alpha, \beta\}) + \sum_{b \in B'} c(\{\beta, b\}) + \max_{b \in B'} c(\{b, l_b\}) .$$

Eq. (1) implies that both expressions are not greater than the former search cost of a' given in Eq. (2), which means that the modification results in an optimal search strategy.

After the latter modification, we can reestablish properties (b) and (c). As this only involves interchanging neighboring queries, the sets of queries before and after the one to $\{\alpha, \beta\}$ are not altered. We repeatedly perform rounds of making the query to $\{\alpha, \beta\}$ earlier, and reestablishing (b) and (c). In each round, the number of queries made before the one to $\{\alpha, \beta\}$ becomes strictly smaller. After a finite number of rounds, property (a) is satisfied, possibly because $\{\alpha, \beta\}$ has become the very first query of the strategy. □

From Lemma 6 one can straightforwardly derive an optimal algorithm. First, sort the inner edges by the cost of their adjacent outer edges. For $j = 0, \ldots, |A| + |B|$, evaluate the search strategy which queries the first j inner edges, then performs a query to $\{\alpha, \beta\}$, then queries the remaining elements of A and B in two different branches of the search tree, but according to the same order. Finally, choose the best among the $|A| + |B| + 1$ evaluated solutions.

The initial sorting step takes time $O(nlogn)$. The rest of the algorithm is similar to merging two sorted sequences and can be implemented such that it runs in linear time.

Theorem 2. *The problem of determining an optimal search strategy for trees of diameter at most 5 admits a polynomial time algorithm.*

Proof. We have given an algorithm for trees having diameter exactly 5. However, trees with diameter less than 5 can be reduced to diameter 5 trees by adding vertices that are connected to the original vertices via cost 0 edges. □

4. A quadratic time algorithm for path instances

In this section we consider the particular case when the tree T is a simple path $P = e_1, \ldots, e_n$, with n edges. A natural dynamic programming procedure finds the optimal decision tree for the path in $O(n^3)$ time. It is based on the observation that the cost of the optimal decision tree $OPT[i, j]$ for the subpath $P[i, j] = e_i, \ldots, e_j$ can be determined as

$$OPT[i, j] = \min_{k=i \ldots j} \big(c(e_k) + \max\{OPT[i, k-1], OPT[k+1, j]\} \big) \tag{3}$$

for $j > i$, and otherwise $OPT[i, i] = c(e_i)$ and $OPT[i, i-1] = 0$. This equation leads to an $O(n^3)$ time algorithm because there are $O(n^2)$ subproblems, and for each subproblem one has to compare $O(n)$ different possibilities for index k. We shall now present a dynamic programming algorithm which cuts a factor of n from the natural DP.

The monotonicity principle/quadrangle inequality [11,19] is a standard trick to speed up dynamic programs of the same flavor — unfortunately, it does not hold here. The monotonicity principle states that if there is an optimal decision tree with root e_r for the path $e_1, \ldots, e_k$, then there is an optimal tree with root $e_{r'}$, with $r' \geq r$, for the path $e_1, \ldots, e_k, e_{k+1}$. To see that it does not hold, consider the path $P_1 = e_1, e_2, e_3, e_4, e_5, e_6$, where $c(e_1) = 1999$, $c(e_2) = 2$, $c(e_3) = 3$, $c(e_4) = c(e_5) = c(e_6) = 1000$. It is not hard to see that the optimal decision tree is rooted at e_3. However, if we consider the extension of P_1 with an edge e_7 of weight $c(e_7) = 3$, the root of the optimal decision tree is e_2.

Assume that we want to compute $OPT[i, j]$ using Eq. (3). We need the previously computed values of $OPT[i, i]$, $OPT[i, i+1], \ldots, OPT[i, j-1]$, and we need the values $OPT[i+1, j]$, $OPT[i+2, j] \ldots, OPT[j, j]$. Straightforward argumentation (or the use of Lemma 8, in the next section) shows that the first sequence is nondecreasing and the second one is non-increasing.

Therefore, for $j > i$, let b_{ij} be the smallest integer s in $[i, j]$ such that $OPT[i, s-1] \geq OPT[s+1, j]$. Note that b_{ij} is well defined because $OPT[i, j-1] \geq OPT[j+1, j] = 0$. In addition, the monotonicity of $OPT[i, \cdot]$ and $OPT[\cdot, j]$ mentioned in the preceding paragraph implies that $OPT[i, k-1] < OPT[k+1, j]$ for each $k \in [i, b_{ij} - 1]$ and $OPT[i, k-1] \geq OPT[k+1, j]$ for each $k \in [b_{ij}, j]$. We call b_{ij} the *transition index* of the interval $[i, j]$.

Let $LC(i,k) := c(e_k) + OPT[i,k-1]$ the *left cost* of k with respect to i. Analogously, we let $RC(k,j) := c(e_k) + OPT[k+1,j]$ and we call it the *right cost* of k w.r.t. j. Exploiting the transition index, Eq. (3) can be rewritten as

$$OPT[i,j] = \min\left\{ \min_{k=i,\ldots,b_{ij}-1} RC(k,j),\ \min_{k=b_{ij},\ldots,j} LC(i,k) \right\}.$$

Motivated by this formula, we let

$$\ell_{ij} = \operatorname*{argmin}_{k\in[b_{ij},j]}\{LC(i,k)\} \quad \text{and} \quad r_{ij} = \operatorname*{argmin}_{k\in[i,b_{ij}-1]}\{RC(k,j)\},$$

so that $OPT[i,j] = \min\{LC(i,\ell_{ij}), RC(r_{ij},j)\}$. Thus, we can find the cost of a minimum cost decision tree for $P[1,n]$ through the following simple procedure.

```
Algorithm PathOPT(P, c, n)
For i = 1, ..., n do
    OPT(i, i − 1) ← 0 ; OPT(i, i) ← c(e_i) ; b_{i,i} ← i
For len = 2, ..., n do
    For i = 1...(n − len + 1) do
        j ← (i + len − 1)
        b_ij ← FindTransitionIndex(i, j)
        ℓ_ij ← FindLeftIndex(i, j)
        r_ij ← FindRightIndex(i, j)
        OPT(i, j) ← min{LC(i, ℓ_ij), RC(r_ij, j)}.
    End do
End do
```

The procedure finds the cost of the optimal decision trees for subpaths of length 2, then for subpaths of length 3 and so on. To show that this new algorithm runs in $O(n^2)$ time we shall explain how to find b_{ij}, ℓ_{ij} and r_{ij} in $O(1)$ amortized time. We start with b_{ij}. The following monotonicity property turns out to be useful.

Lemma 7. *Let i,j be such that $1 \le i < j \le n$ and $j \ge i+2$. Then, $b_{i,j-1} \le b_{ij} \le b_{i+1,j}$.*

Proof. We only show $b_{i,j-1} \le b_{ij}$, because the other inequality can be shown symmetrically. By the definition of $b_{i,j-1}$, we have $OPT[i,k-1] < OPT[k+1,j-1]$ for each $k \in [i, b_{i,j-1}-1]$. By Lemma 8 it holds that $OPT[k+1,j-1] \le OPT[k+1,j]$ for any k. It follows that $OPT[i,k-1] < OPT[k+1,j]$ for $k \in [i, b_{i,j-1}-1]$, which is why b_{ij} cannot be smaller than $b_{i,j-1}$. □

The above lemma implies that for computing b_{ij} it suffices to consider the positions between $b_{i,j-1}$ and $b_{i+1,j}$. Among these positions, b_{ij} is computed as the smallest index k such that $LC(i,k) \ge RC(k,j)$, where all values of LC and RC required for this computation can be determined reusing previously computed values of $OPT[\cdot,\cdot]$. The pseudo-code for FindTransitionIndex() is presented below.

```
Procedure FindTransitionIndex(i, j)
For s = b_{i,j−1}, ..., b_{i+1,j} do
    If OPT[i, s − 1] ≥ OPT[s + 1, j]
        Break
End do
Return s
```

To calculate the overall time spent by procedure FindTransitionIndex(), fix a value of *len* (algorithm's outer loop); the total number of positions that are considered for determining the indices b_{ij}, with $j - i = len - 1$, is

$$\sum_{i=1}^{n-len+1} (b_{i+1,i+len-1} - b_{i,i+len-2} + 1) = b_{n-len+2,n} - b_{1,len-1} + (n - len + 1) < 2n.$$

By considering all possible values for *len*, we conclude that the algorithm spends $O(n^2)$ to find all b_{ij}'s.

It remains to show how to find the indices ℓ_{ij} and r_{ij} in $O(1)$ amortized time. We just detail how to compute ℓ_{ij} because r_{ij} can be computed in a symmetric way.

Assume some i to be fixed throughout the following paragraphs. In order to find the ℓ_{ij}'s in an efficient way, we organize candidates for ℓ_{ij} in a linear list L_i. Recall that the candidates for ℓ_{ij} are the indices $b_{ij},\ldots,j$. If $k > k'$ and $LC(i,k') \ge LC(i,k)$ we say that k' is *left dominated* by k with respect to i. Note that if k' is left dominated by some k then we can conclude that $\ell_{ij} \ne k'$ for $j = k,\ldots,n$ since k is a choice better than k' (whenever k' is a candidate). This is a key property for finding the ℓ_{ij}'s efficiently because it allows to discard some candidates from L_i during the algorithm's execution. The pseudo-code for finding ℓ_{ij} is presented below.

```
Procedure FindLeftIndex(i, j)
While the head of L_i is smaller than b_ij do
    Remove the element at the head of L_i
End do
While the left cost of the tail of L_i is larger than LC(i, j) do
    Remove the element at the tail of L_i
End do
Insert j at the tail of L_i
Return the head of L_i
```

The structure L_i can be implemented as a linked list. It shall be easy to observe that L_i is simultaneously sorted by increasing order of edge indices and by increasing order of left costs w.r.t. i. In the first While loop we remove from L_i every index smaller than b_{ij}. This is done because, as b_{ij} is nondecreasing in j, these indices can never be candidate indices any more. In the second While loop we remove from L_i all indices k such that $LC(i, j) \leq LC(i, k)$. This is because these indices are left dominated by j.

To calculate the overall time spent by procedure FindLeftIndex(), fix a value of i. Clearly, the overall time spent by this procedure, when the first argument is i, is proportional to the number of elements inserted at L_i, which is at most $n - i$. Thus, the overall cost of FindLeftIndex() is $O(n^2)$.

The pseudo-code for FindRightIndex is presented below. It uses a structure R_j, which is analogous to L_i, and it is overall execution time is also $O(n^2)$.

```
Procedure FindRightIndex(i, j)
While the head of R_j is larger than or equal to b_ij do
    Remove the element at the head of R_j
End do
While the right cost of the tail of R_j is larger than RC(i, j) do
    Remove the element at the tail of R_j
End do
Insert i at the tail of R_j
Return the head of R_j
```

In summary, we have the following theorem.

Theorem 3. *There is an $O(n^2)$ time algorithm that finds the optimal search tree for the problem of searching in a weighted path.*

5. An $O(\log n / \log\log\log n)$ approximation algorithm

In this section, we present an $O(\log n / \log\log\log n)$ approximation algorithm for general weighted trees.

We shall first present two natural lower bounds on the cost of the optimal decision tree for a given weighted tree. We shall use $OPT(T, \mathbf{c})$ to denote the cost of an optimal decision tree for the weighted tree instance $(T, \mathbf{c})$. When the cost assignment is clear from the context we use $OPT(T)$ instead of $OPT(T, \mathbf{c})$.

Lemma 8. *Let T' be a subtree of T. Then, $OPT(T, \mathbf{c}) \geq OPT(T', \mathbf{c})$.*

Proof. Let D be a decision tree for the instance $(T, \mathbf{c})$. It is not hard to see that, given a subtree T' of T we can turn D into a decision tree for the instance $(T', \mathbf{c})$ by repeatedly performing the following transformation: let $e = \{u, v\}$ be an edge of T such that $e \notin T'$. Let T_u and T_v be the connected components of $T - e$, and assume, w.l.o.g., that T' is a subtree of T_u. Let v_e be the node in D labeled by e and D_u be the subtree rooted at the child of v_e which is the decision tree for some subtree of T_u. Then we substitute in D the subtree rooted at v_e with D_u.

Let D' be the resulting decision tree after having performed the above transformation on each edge not in T'. Clearly, it holds that $cost(D') \leq cost(D)$. Hence $OPT(T') \leq cost(D') \leq cost(D)$. □

Lemma 9. *Let T' be a subtree of T. Then, $OPT(T, \mathbf{c}) \geq c_{min}(T') \log |T'|$, where $c_{\min}(T')$ is the minimum cost of an edge of T' according to the cost assignment $\mathbf{c}$.*

Proof. Let D' be the optimal decision tree for a subtree T' of T. Then D' has $|T'|$ leaves, where $|T'|$ denotes the number of vertices in T'. Hence the number of nodes in some root-to-leaf path in D' is not smaller than $\log |T'|$. Since any edge in T' has weight at least $c_{\min}(T')$, it follows that the sum of the edge costs on some root-to-leaf path in D' is not smaller than $c_{\min}(T') \log |T'|$, and this is a lower bound on the cost of D'. Therefore, we have $OPT(T', \mathbf{c}) = cost(D') \geq c_{\min}(T') \log |T'|$. This together with Lemma 8 provides the bound in the statement. □

The following algorithmic result will be employed to solve small instances of the problem.

Proposition 1. *Let T be a weighted tree. An optimal decision tree for T can be constructed in $O(2^{|T|} \cdot |T|)$ time.*

Proof. Let E be the set of edges of T. We have that

$$OPT(T) = \min_{e=\{u,v\}\in E} (c(e) + \max\{OPT(T_u), OPT(T_v)\}),$$

where T_u (T_v) is the tree of $T - e$ that contains u (v). Since there are at most $2^{|T|}$ subtrees in T and there are at most $|T|$ choices for the root of T, it follows that this equation can be solved in $O(|T| \times 2^{|T|})$ time by means of dynamic programming. The optimal decision tree can be easily computed from the values of $OPT(\cdot)$. □

5.1. Algorithm's description

For a forest F we denote by $\mathcal{K}(F)$ the set of connected components (trees) in F. We use $K(F)$ to denote the size of the largest component of F, i.e., $K(F) = \max_{C\in\mathcal{K}(F)} |C|$.

Let $t \in (0, 1/2)$ be a parameter whose exact value will be determined in the course of the analysis. Given a weighted tree T, the algorithm makes use of a subset S of vertices in T that satisfies the following properties: (i) S is connected in T (it induces a subtree in T); (ii) all edges of the subtree $T[S]$, induced by S on T, are reasonably good separators for T, that is, $K(T - e) \leq (1 - t)|T|$ for every $e \in T[S]$; (iii) each component of $\mathcal{K}(T - S)$ has size at most $t \cdot |T|$. We refer to S as the separator set of T.

The set S can be constructed as follows: we start with a centroid of T (i.e., a vertex v with $K(T - v) \leq |T|/2$) and then we keep on adding vertices v such that v is adjacent to some vertex $u \in S$ for which $K(T - \{u, v\}) \leq (1 - t)|T|$. This procedure stops when there is no vertex in $T - S$ which satisfies the required conditions. By construction, the set S satisfies properties (i) and (ii). The next proposition assures that it also satisfies the third property.

Proposition 2. *Each component of $\mathcal{K}(T - S)$ has size at most $t \cdot |T|$.*

Proof. Let C be a component of $\mathcal{K}(T - S)$ and let e be the edge that connects C to $T[S]$. In addition, let u be the endpoint of e that belongs to C. Since u is not added to S, the larger component of $T - e$ has size larger than $(1 - t)|T|$, hence the smaller component of $T - e$ has size smaller than or equal to $t|T|$. This smaller component must be C, for otherwise S would not contain a centroid vertex, which is not possible. □

Let n be the number of vertices of the input tree. The algorithm works in a recursive way. In a generic call, it receives a weighted tree T, which is a subtree of the input tree. If the size of T is smaller than some constant, it constructs the optimal tree for T by brute force. Otherwise, it constructs the separator set S for T and then, depending on the size of S, proceeds in accordance with one of the following two cases:

Case 1: $|S| > \log n$. Let $e^* = \{u, v\}$ be the edge of minimum cost in $T[S]$. Let T_u (resp. T_v) be the connected component of $T - e^*$ containing u (resp v). The algorithm probes the edge e^* and then recurses in the subtrees T_u and T_v. The choice of e^* guarantees that the algorithm probes an edge which is both cheap and a reasonably good separator.

Let $APP(T, n)$ denote the approximation ratio achieved by the algorithm on a subtree T of an input tree with n vertices. This construction yields the following equation

$$APP(T, n) \leq \frac{c(e^*) + \max\{cost(T_v), cost(T_u)\}}{OPT(T)},$$

where $cost(T_v)$ (resp. ($cost(T_u)$) is the cost of the decision tree constructed by the algorithm for input tree T_v (resp. (T_u)).

Since $|S| > \log n$ it follows from Lemma 9 that $OPT(T) \geq c(e^*) \log\log n$. In addition, it follows from Lemma 8 that $OPT(T) \geq \max\{OPT(T_v), OPT(T_u)\}$. Thus,

$$APP(T, n) \leq \frac{c(e^*) + \max\{cost(T_v), cost(T_u)\}}{\max\{c(e^*) \log\log n, OPT(T_v), OPT(T_u)\}} \leq \frac{1}{\log\log n} + APP(T', n) \tag{4}$$

where T' is a subtree of T with at most $(1 - t) \cdot |T|$ vertices.

Case 2: $|S| \leq \log n$. In this case the algorithm takes advantage of the fact that S is a good separator for T (Proposition 2) and that an optimal decision tree for $T[S]$ can be constructed in $O(n \log n)$ as shown by Proposition 1. Let S^T be the vertices of S that are adjacent to some vertex in $T - S$. The algorithm proceeds as follows:

1. Build an optimal decision tree for $T[S]$;
2. For each $v \in S^T$ build the optimal decision tree for S_v, where S_v is the star induced by v and the vertices of $T - S$ adjacent to v.
3. Recurse in the components of $\mathcal{K}(T - S)$;
4. Assemble a decision tree D for T as follows:
   (a) For each $v \in S^T$, replace the leaf of the optimal tree for $T[S]$ corresponding to v with the optimal decision tree for S_v;
   (b) For each $v \in S^T$ and for each $w \in S_v - \{v\}$, replace the leaf of the optimal decision tree for S_v corresponding to w with the decision tree recursively constructed for the component of $T - S$ that contains w.

Note that any decision tree for a star is optimal. This construction yields the following estimate of the approximation achieved by the algorithm:

$$APP(T, n) \leq \frac{OPT(T[S]) + \max_{v \in S^T}\{OPT(S_v)\} + \max_{C \in \mathcal{K}(T-S)}\{cost(C)\}}{OPT(T)} \leq 2 + APP(T', n) \tag{5}$$

where T' is a subtree of T with at most $t|T|$ vertices.

To give an upper bound on $APP(T, n)$, we apply repeatedly Eqs. (4) and (5), depending whether we reach Cases 1 or Case 2. Thus, we get that

$$APP(T, n) \leq \sum_{i=1}^{k_1} \left(\frac{1}{\log\log n}\right) + \sum_{i=1}^{k_2} 2 = \frac{k_1}{\log\log n} + 2k_2$$

where k_1 and k_2 are, respectively, the number of times Cases 1 and 2 are reached in the selected path of the algorithm's execution tree.

To obtain an upper bound on k_1, we observe that whenever the algorithm reaches Case 1, the size of the current tree is reduced by a factor of $(1 - t)$. By using the well known inequality $(1 - t)^{1/t} \leq 1/e$, we conclude that the size of the tree is reduced by a factor of at least $1/e$ after the algorithm reaches Case 1 $1/t$ times. Thus, the size of the tree is reduced by a factor of $1/n$ if the Case 1 is reached $\ln n/t$ times. Therefore, $k_1 \leq \ln n/t$.

On the other hand, whenever the algorithm reaches Case 2, the size of the tree is reduced by a factor of t, so that Case 2 can be reached at most $\log n/\log(1/t)$ times before the size of the tree gets smaller than a constant. It follows that

$$APP(T, n) \leq \frac{k_1}{\log\log n} + 2k_2 \leq \frac{\ln n}{t\log\log n} + \frac{\log n}{\log(1/t)}.$$

By setting $t = \log\log\log n/\log\log n$, we obtain that $APP(T, n)$ is

$$O\left(\frac{\log n}{\log\log\log n} + \frac{\log n}{\log\log\log n - \log\log\log\log n}\right) = O\left(\frac{\log n}{\log\log\log n}\right).$$

To determine an upper bound on the algorithm's running time we first note that the set S can be found in $O(n\log n)$ time. Furthermore, all steps in both cases 1 and 2 can be also implemented in $O(n\log n)$ time. Thus, the time spent in a recursive call is $O(n\log n)$. Since the algorithm executes at most n calls, it follows that the algorithm runs in $O(n^2\log n)$ time. Summarizing, we have shown the following result.

Theorem 4. *There is an $O(\log n/\log\log\log n)$-approximation algorithm for the problem of searching in weighted trees that runs in $O(n^2\log n)$ time.*

We shall remark that the threshold $t = \log\log\log n/\log\log n$, used to decide whether the algorithm shall act in accordance with Case 1 or Case 2, is fixed throughout the algorithm.

6. Conclusions

In this paper we considered the problem of searching a node in a tree via edge queries of non-uniform weights. This problem is a particular case of the Binary Identification Problem (BIP) where the underlying space of objects and tests can be represented by a weighted tree. In addition, this problem is equivalent to the weighted version of the edge ranking problem.

We gave a complete characterization of the computational complexity of the problem in terms of the diameter and the maximum degree of a tree. With respect to the diameter, we proved that the class of trees of diameter at most 5 is polynomially solvable while the class of tree of diameter at most 6 is NP-Hard. With regards to the maximum degree, the class of trees with degree at most 2 admits a natural $O(n^3)$ time dynamic programming algorithm — we established here that the class of trees of degree at most 3 is NP-Hard. In addition, we presented an $O(n^2)$ time algorithm for the class of trees of maximum degree 2 (paths) improving upon the natural $O(n^3)$ time algorithm.

Finally, we presented an $o(\log n)$ approximation algorithm for unrestricted trees. This last result suggests that the BIP for trees with non-uniform weights on the tests (queries) is easier from a computational complexity perspective than the general BIP since the latter does not admit an $o(\log n)$ approximation unless $P = NP$. A main question which remains open regards the existence of a constant factor approximation for general trees.

Acknowledgment

We thank Marco Molinaro for several inspiring discussions. The first author was partially supported by the German Academic Exchange Service (DAAD) grant, ref. code A/11/15927. The second author's work was supported by a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD).

References

[1] Y. Ben-Asher, E. Farchi, I. Newman, Optimal search in trees, SIAM J. Comput. 28 (6) (1999) 2090–2102.
[2] M. Charikar, R. Fagin, V. Guruswami, J.M. Kleinberg, P. Raghavan, A. Sahai, Query strategies for priced information, J. Comput. System Sci. 64 (4) (2002) 785–819.
[3] F. Cicalese, T. Jacobs, E. Laber, M. Molinaro, On Greedy algorithms for decision trees, in: Proc. of ISAAC 2010, in: LNCS, 6507, 2010, pp. 206–217.
[4] F. Cicalese, T. Jacobs, E. Laber, M. Molinaro, On the complexity of searching in trees: average-case minimization, in: Proc. of ICALP 2010, in: LNCS, 6198, 2010, pp. 527–539.
[5] P. de la Torre, R. Greenlaw, A. Schäffer, Optimal edge ranking of trees in polynomial time, Algorithmica 13 (6) (1995) 592–618.
[6] D. Dereniowski, Edge ranking of weighted trees, DAM 154 (2006) 1198–1209.
[7] D. Dereniowski, Edge ranking and searching in partial orders, DAM 156 (2008) 2493–2500.
[8] M. Garey, Optimal binary identification procedures, SIAM J. Appl. Math. 23 (2) (1972) 173–186.
[9] A. Iyer, H. Ratliff, G. Vijayan, On an edge ranking problem of trees and graphs, Discrete Appl. Math. 30 (1) (1991) 43–52.
[10] A.V. Iyer, H.D. Ratliff, G. Vijayan, On an edge ranking problem of trees and graphs, Discrete Appl. Math. 30 (1991) 43–52.
[11] D. Knuth, Optimum binary search trees, Acta. Informat. 1 (1971) 14–25.
[12] E. Laber, R. Milidiú, A. Pessoa, On binary searching with non-uniform costs, in: Proc. of SODA'01, 2001, pp. 855–864.
[13] E. Laber, L. Nogueira, On the hardness of the minimum height decision tree problem, Discrete Appl. Math. 144 (1-2) (2004) 209–212.
[14] T.W. Lam, F.L. Yue, Optimal edge ranking of trees in linear time, in: Proc. of SODA'98, 1998, pp. 436–445.
[15] M. Lipman, J. Abrahams, Minimum average cost testing for partially ordered components, IEEE Trans. Inform. Theory 41 (1) (1995) 287–291.
[16] K. Makino, Y. Uno, T. Ibaraki, On minimum edge ranking spanning trees, J. Algorithms 38 (2001) 411–437.
[17] S. Mozes, K. Onak, O. Weimann, Finding an optimal tree searching strategy in linear time, in: Proc. of SODA'08, 2008, pp. 1096–1105.
[18] K. Onak, P. Parys, Generalization of binary search: Searching in trees and forest-like partial orders, in: Proc. of FOCS'06, 2006, pp. 379–388.
[19] F.F. Yao, Efficient dynamic programming using quadrangle inequalities, in: Proc. of STOC'80, 1980, pp. 429–435.
[20] L. Hyafil, R. Rivest, Constructing optimal binary decision trees is NP-complete, Information Processing Letters 5 (1) (1976) 15–17.
[21] E.M. Arkin, H. Meijer, J.S.B. Mitchell, D. Rappaport, S.S. Skiena, Decision trees for geometric models, Internat. J. Comput. Geom. Appl. 8 (3) (1998) 343–364.